

Introduzione al linguaggio C

Puntatori

Violetta Lonati

Università degli studi di Milano
Dipartimento di Informatica

Laboratorio di algoritmi e strutture dati
Corso di laurea in Informatica

19 ottobre 2017

Argomenti

Puntatori

Operatori & e *

Puntatori come argomento di funzione

Puntatori come valori restituiti

Puntatori a strutture

Puntatori e array

Stringhe e puntatori

Variabili di tipo puntatore

Indirizzi di memoria

La memoria è divisa in byte, ognuno dei quali ha un indirizzo. Ogni variabile occupa uno o più byte a seconda del suo tipo.

Variabili puntatore

- ▶ Sono variabili che hanno come **valore** un indirizzo di memoria.
- ▶ Se **p** contiene l'indirizzo di memoria in cui si trova la variabile **i**, diciamo che **p punta a i**.

Dichiarazione di variabili puntatore

Dato che due variabili di tipo diverso occupano quantità di memoria diversa, è importante specificare che tipo di variabile può puntare un puntatore:

```
int *p;    /* p punta a variabili di tipo int */
int a, b, *p, n[10]; /* dichiarazione composta*/
```

Operatori di indirizzo (&) e indirezione (*)

- ▶ L'operatore & consente di ottenere l'indirizzo di memoria di una variabile:

```
int i, *p; /* dichiaro il puntatore a intero p */  
p = &i; /* assegno a p l'indirizzo di i */
```

- ▶ E' possibile inizializzare una variabile puntatore in fase di dichiarazione:

```
int i;  
int *p = &i;
```

- ▶ L'operatore * consente di accedere alla variabile puntata dal puntatore:

```
int i = 3, *p = &i;  
printf( "%d", *p ); /* stampa 3 */
```

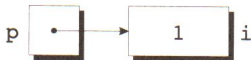
- ▶ Cambiando il valore di *p si cambia il valore della variabile puntata!

Esempio

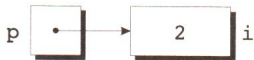
```
p = &i;
```



```
i = 1;
```



```
printf(“%d\n”, i);      /* stampa 1 */  
printf(“%d\n”, *p);    /* stampa 1 */  
*p = 2;
```



```
printf(“%d\n”, i);      /* stampa 2 */  
printf(“%d\n”, *p);    /* stampa 2 */
```

Assegnamento di puntatori

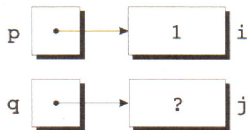
- ▶ Attenzione! Se un puntatore `p` non è inizializzato, il suo valore non è definito, quindi non è definito nemmeno `*p`; `p` potrebbe puntare ad uno spazio di memoria qualsiasi, anche riservato al sistema operativo → segmentation fault!
- ▶ La seguente porzione di programma copia il valore di `p` (cioè l'indirizzo di `i`) dentro `q`, ovvero dopo la copia anche `q` punterà alla variabile `i`.

```
int i, j, *p, *q;  
p = &i;  
q = p; /* copia di puntatori */
```

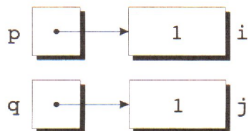
- ▶ Cambiando il valore di `*p` si cambia automaticamente anche il valore di `*q`.
- ▶ Fate attenzione a non confondere `q = p;` con `*q = *p;`: nel secondo caso, il valore della variabile puntata da `p` viene assegnato alla variabile puntata da `q`.

Esempio

```
p = &i;  
q = &j;  
i = 1;
```



```
*q = *p;
```



Puntatori come argomento di funzione

- ▶ Passando ad una funzione un puntatore ad una variabile, si può fare in modo che la funzione modifichi il valore della variabile stessa. Invece di passare la variabile `x` dovremo passare come argomento il suo indirizzo, ovvero `&x`.
- ▶ Al momento della chiamata il parametro `p` (di tipo puntatore) corrispondente a `&x` verrà inizializzato col valore di `&x` ovvero con l'indirizzo di `x`.

Esempio: l'uso di `scanf`

Passo alla funzione `scanf` l'indirizzo della variabile `i` di cui voglio cambiare il valore attraverso la chiamata della funzione `scanf` stessa:

```
int i;  
scanf( "%d", &i );
```

L'operatore `&` nella `scanf` non è richiesto se come argomento uso un punt.:

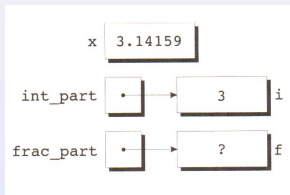
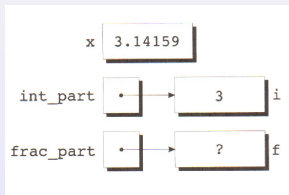
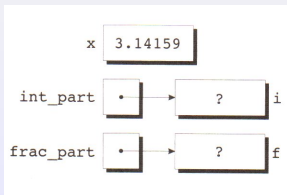
```
int i, *p = &i;  
scanf( "%d", p );
```


Esempio

```
void decompose( float x, int *int_part,
               float *frac_part) {
    *int_part = (int) x;
    *frac_part = x - *int_part;
}
```

Se `i` e `f` sono rispettivamente di tipo `int` e `float`, possiamo effettuare la chiamata

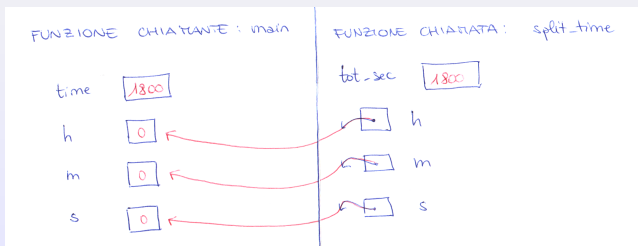
```
decompose( 3.14159, &i, &f);
```



Esempio

```
void split_time( long int tot_sec, int *h,
                int *m, int *s ) {
    *h = tot_sec / 3600;          tot_sec %= 3600;
    *m = tot_sec / 60;           *s = tot_sec % 60;
}

int main( void ) {
    long int time = 1800;
    int h=0, m=0, s=0;
    split_time( time, &h, &m, &s );
    printf( "h=%d, m=%d, s=%d\n", h, m, s );
    return 0;
}
```



Puntatori come valori restituiti

E' possibile scrivere funzioni che restituiscono puntatori. Ad esempio:

```
int *max( int *a, int *b) {
    if ( *a > *b )
        return a;
    else
        return b;
}
```

Per invocare la funzione `max`, le passiamo due puntatori a variabili `int` e salviamo il risultato in una variabile puntatore:

```
int *p, i, j;
...
p = max( &i, &j);
```

Attenzione: non restituite mai un puntatore ad una variabile locale (a meno di averla dichiarata `static`)!

Puntatori a strutture

Accedere ai membri di una struttura usando un puntatore è un'operazione molto frequente, tanto che il linguaggio C fornisce l'operatore `->` specifico per questo scopo:

```
typedef struct {
    float x, y;
} punto;

typedef struct {
    punto p1, p2;
} rettangolo;

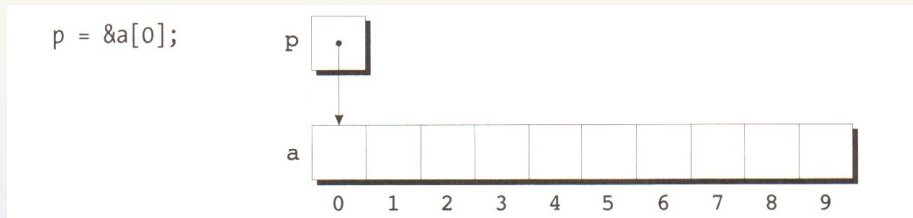
/* stampa i vertici che def. il rett. puntato da r*/
void stampa( rettangolo *r ) {
    printf( "Rett. di vertici (%f, %f) e (%f, %f).\n",
           r -> p1.x, r -> p1.y,
           r -> p2.x, r -> p2.y );
}
```

Aritmetica dei puntatori

Dichiarati

```
int a[10], *p;
```

possiamo fare in modo che un puntatore `p` punti ad `a[0]`:

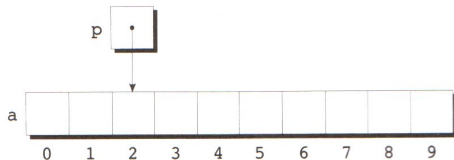


Usando i puntatori, possiamo anche accedere agli altri elementi di `a` usando l'**aritmetica dei puntatori** che prevede 3 operazioni:

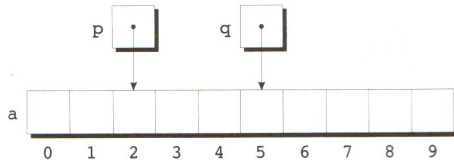
- ▶ sommare un intero a un puntatore;
- ▶ sottrarre un intero a un puntatore;
- ▶ sottrarre da un puntatore un altro puntatore.

Sommare un intero a un puntatore

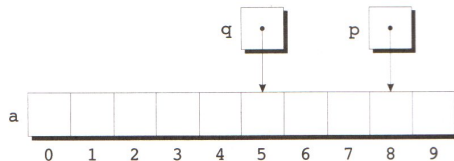
```
p = &a[2];
```



```
q = p + 3;
```

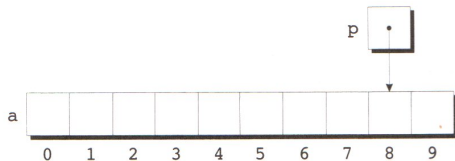


```
p += 6;
```

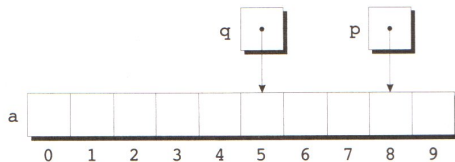


Sottrarre un intero a un puntatore

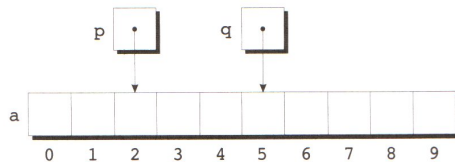
```
p = &a[8];
```



```
q = p - 3;
```



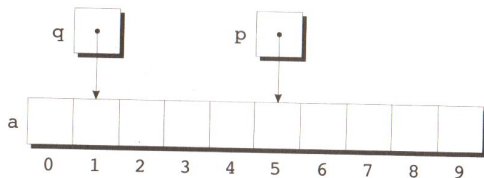
```
p -= 6;
```



Sottrarre da un puntatore un altro puntatore

```
p = &a[5];  
q = &a[1];
```

```
i = p - q; /* i è uguale a 4 */  
i = q - p; /* i è uguale a -4 */
```



Puntatori e array

Uso di puntatori per scorrere array

```
int a[N], *p, sum = 0;

for ( p = &a[0]; p < &a[N]; p++ )
    sum += *p;
```

Combinazione tra * e ++

- ▶ `*p++` equivale a `*(p++)`: prende il valore dell'oggetto puntato da `p`, poi incrementa il puntatore;
- ▶ `(*p)++`: prende il valore dell'oggetto puntato da `p`, poi incrementa tale valore;
- ▶ `+++p`: incrementa `p`, poi prende il valore dell'oggetto puntato;
- ▶ `++*p`: incrementa il valore dell'oggetto puntato e prende il valore incrementato

Nomi di array come puntatori costanti

Il nome di un array può essere usato come puntatore costante al primo elemento dell'array. `a[i]` corrisponde a `*(a + i)`.

```
int a[N];
*a = 7;           /* salva 7 in a[0] */
*(a + 1) = 12    /* salva 12 in a[1] */
```

Uso di puntatori per scorrere array (rivisitato)

```
int a[N], *p, sum = 0;
for ( p = a; p < a + N; p++ )
    sum += *p;
```

Attenzione: non posso cambiare il valore di un array!

```
a++;           /* SBAGLIATO! */
a = p;        /* SBAGLIATO! */
```

Array come argomenti di funzioni

Il nome di un array argomento di funzione è sempre considerato come un puntatore.

- ▶ l'array non viene copiato (maggiore efficienza);
- ▶ l'array non è protetto da cambiamento (usare `const`);
- ▶ non c'è modo di sapere quanto è lungo l'array;
- ▶ è possibile passare porzioni di array:

```
somma( a, n );  
somma( &a[5], n );
```

- ▶ il parametro può essere definito indifferentemente
 - ▶ come array `int a[]`
 - ▶ o come puntatore `int *a`

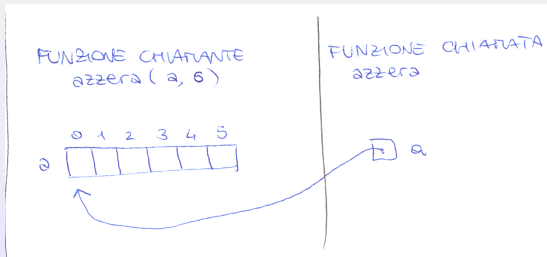
Attenzione: nelle dichiarazioni di variabili invece c'è differenza tra queste due dichiarazioni:

```
int a[N];      /* alloca lo spazio per N interi */  
int *a;       /* alloca lo spazio solo per  
              un puntatore a intero */
```

Esempio

```
void azzera( int a[], int n) {  
    int i;  
    for ( i = 0; i < n; i++ )  
        a[i] = 0;  
}
```

Il vettore `a` viene trattato come puntatore, quindi non viene copiato, ma viene passato alla funzione l'indirizzo del suo primo elemento `&a[0]`. La funzione `azzera` quindi direttamente gli elementi di `a` e non di una copia locale!



Array come argomenti di funzione

- ▶ Se il vettore è unidimensionale, la dimensione può essere omessa.

```
int f( int a[] ) { ... }
```

- ▶ Se il vettore è multidimensionale, solo la prima dimensione può essere omessa.

```
int f( int a[][LUN] ) { ... }
```

- ▶ la funzione non ha modo di sapere quanto è lungo il vettore (l'operatore `sizeof` non può essere usato...) quindi può essere utile passare la lunghezza come parametro aggiuntivo:

```
int somma_array( int a[], int n ) {  
    int i, sum = 0;  
    for ( i = 0; i < n; i++ )  
        sum += a[i];  
    return sum;  
}
```

Stringhe e puntatori

- ▶ Ogni letterale stringa (es: "ciao") è un array di caratteri, quindi è trattato come puntatore a carattere.
- ▶ Ad esempio il primo parametro nel prototipo della funzione printf è un puntatore a carattere:

```
int printf( const char *format, ...);
```

- ▶ Se una stringa è argomento di una funzione, alla chiamata essa non viene copiata, ma viene passato l'indirizzo della sua prima lettera.

Stringhe e puntatori - attenzione alle dichiarazioni

Dichiarazione come array

```
char data[] = "13_maggio";
```

- ▶ data è un vettore che contiene i caratteri '1', '3', ...
- ▶ i singoli caratteri possono essere modificati (Es: `data[1] = '4'`);

Dichiarazione come puntatore

```
char *data = "13_maggio";
```

- ▶ il letterale costante "13_maggio" è memorizzato in un array;
- ▶ l'inizializzazione fa sì che `data` punti al letterale costante;
- ▶ il puntatore `data` può essere modificato in modo che punti altrove.

File di intestazione `string.h`

Copia di stringhe

Non si possono usare assegnamenti tipo `str = "abcd"`; usiamo la funzione

```
char *strcpy(char *dest, const char *src);
```

che copia `src` in `dest` e ne restituisce l'indirizzo.

Esempio: `strcpy(str, "abcd")` copia `"abcd"` in `str`.

Concatenazione di stringhe

```
char *strcat(char *dest, const char *src);
```

aggiunge il contenuto di `src` alla fine di `dest` e restituisce `dest` (ovvero il puntatore alla stringa risultante).

Confronto tra stringhe

```
int strcmp(const char *s1, const char *s2);
```

restituisce un valore maggiore, uguale o minore di 0 a seconda che `s1` sia maggiore, uguale o minore di `s2`.

Esempio: calcolare la lunghezza di una stringa

```
/* Prima versione */
int lun_stringa( const char *s ) {
    int n = 0;
    while ( *s != '\0' ) {
        n++; s++;
    }
    return n;
}
```

```
/* Seconda versione */
int lun_stringa( const char *s ) {
    int n = 0;
    while ( *s++ != '\0' )
        n++;
    return n;
}
```

Esempio: calcolare la lunghezza di una stringa - continua

```
/* Terza versione */
int lun_stringa( const char *s ) {
    int n = 0;
    while ( *s++ )
        n++;
    return n;
}
```

```
/* Quarta versione */
int lun_stringa( const char *s ) {
    const char *p = s;
    while ( *s++ )
        ;
    return s - p - 1;
}
```